Institut de Mathématiques d'Orsay
Université Paris-Saclay

Rapport sur les algorithmes d'approximation et de méta-heuristique pour le problème de l'arbre de Steiner

Éric Aubinais, Farid Najar
Master Mathématiques de l'Intelligence Artificielle

Table des matières

I Contexte

Imaginons que nous avons un réseau avec une source et plusieurs destinations. Nous voulons avoir des chemins depuis la source vers les destinations sans être coupé par d'autres chemins. Dit autrement, nous voulons un graphe sans cycles, i.e. un arbre, en partant de la source et en ayant les destinations qu'on veut visiter. De plus, pour aller d'un point à l'autre faut payer un coût. Le but est de trouver l'arbre qui vérifie les conditions énoncées de coût minimal.

II Modélisation

Nous pouvons modéliser le réseau par un graphe connexe $G = (V, E)$ avec V l'ensemble des sommets et E l'ensembles des arrêtes. On introduit aussi une fonction $w : E \to \mathbb{N}$ qui à chaque arrête $e \in E$ attribut un poids $w(e)$. Soit T l'ensemble des "terminaux" qui sont tout simplement les destinations que nous avons énoncé. On remarque que la source peut être considérée comme un terminal. En effet, comme notre solution est un arbre, on peut le représenter comme on veut en prenant un nœud quelconque comme source. Alors notre problème est de trouver un arbre $A = (V', E')$ (graphe connexe sans cycles), tel que, $T \subseteq V'$ et on veut minimiser $\sum_{e \in E'} w(e)$. On représente les solutions comme une liste de 0 ou 1 de taille $|E|$. En numérotant les arrêtes, pour une solution s, s_i nous dit si le i-ème arrête est dans l'arbre ou pas (1 oui, 0 non).

III Complexité

IV Algorithme d'approximation et taux d'approximation

Comme le problème est NP-Complet, alors si on considère $P \neq NP$, on ne peut construire un algorithme donnant une solution optimale en temps polynomial. Dans ce cas, nous avons plusieurs façon de trouver une solution qui est proche ou, dans certain cas, égal à une solution optimale. Dans ce rapport, nous allons faire et évaluer deux de ces méthodes, "**approximation**" et "**méta-heuristique**". Commençons par l'approximation. Une approximation consiste à essayer, à l'aide de différentes techniques, de trouver la meilleure solution possible en temps polynomial avec une distance maximale déterminée par rapport à la solution optimale qui est appelé le **taux d'approximation** qui est un critère essentiel qui nous permet d'évaluer ce genre d'algorithmes. On peut construire un algorithme d'approximation de différentes manières. Dans cette section, nous allons voir une de ces manières.

IV.1 Algorithme d'approximation

Nous cherchons un arbre qui passe par tous les sommets terminaux. On remarque que trouver un tel arbre dans un graphe quelconque est équivalent à trouver cet arbre dans un graphe complet avec les terminaux comme sommet. En effet, si on construit ce graphe en mettant la distance minimale entre chaque sommet comme poids de l'arrête qui les relie, et trouver un arbre couvrant de poids minimum qui passe par tous les sommets, on est sûr que nous avons un arbre pour le graphe d'origine en passant par les terminaux. On doit aussi enregistrer les plus courts chemins dans la mémoire afin de ne pas être obligé de les recalculer, et ainsi, ajouter une complexité supplémentaire à notre algorithme. Pour récapituler, nous avons le procédé suivant :

1. On construit G_+ le graphe complet qui a les terminaux comme sommets et sur chaque arrête, on met le poids du plus court chemin. On enregistre, à chaque fois, le plus court chemin dans la mémoire. Pour calculer les plus courts chemins, on utilise l'algorithme Dijkstra.
   Complexité : On pose $n = |V|$. Pour Dijkstra, on a une complexité $O(n^2 \log(n))$. Et pour le stockage des chemins, on a $O(n)$

2. On construit A, l'arbre couvrant de poids minimum de $G+$. Cet algorithme a une complexité de $O(|T| \log(|T|))$.

3. On renvoie l'union des arrêtes qui composent les plus courts chemins. Comme on a enregistré ces chemins, on une complexité $O(1)$.

IV.2 Taux d'approximation

Soit T^* une solution optimale. On sait déjà qu'une solution C donnée par le cycle eulérien vérifie $C = 2T^*$. On a solution C' en prenant les sommets par ordre de première apparition et on a $C' \leq C$. Soit A une solution de l'approximation. On a $A \leq C'$, car, on peut supprimer, au minimum, une arrête de C' afin de ne pas avoir de cycles. Alors, on a $A \leq 2T^*$ par transitivité. Comme on renvoie l'union des arrêtes, on a $Resultat \leq A$. On a donc 2-approximation.

V Méta-heuristiques

L'approximation est une bonne façon de trouver les solutions, mais, il n'est pas évident de trouver un algorithme d'approximation. Dans cette section, nous allons voir deux algorithmes, de la famille des algorithmes méta-heuristiques, qui sont probabilistes et donnent des résultats différents à chaque exécution. L'intérêt de ces algorithmes est dans leur flexibilité et la facilité d'implémentation. En effet, contrairement au cas d'approximation, nous n'avons pas besoin de chercher une manière spécifique de trouver la meilleure solution. Dans de nombreux cas, on n'a toujours pas trouver un algorithme d'approximation et dans tant d'autres, les taux d'approximations sont assez élevés. Pour utiliser mes méta-heuristiques, il suffit d'adapter les algorithmes au problème et utiliser le plus pertinent selon les cas.

Les méta-heuristiques partent d'une solution aléatoire et cherchent de nouvelles solutions en transformant la solution de base. Ensuite, elles évaluent ces solutions et gardent la ou les meilleures. Elles répètent cette opération un nombre de fois qu'il faut déterminer. Finalement, elles rendent la meilleure qu'ils ont trouvé.

Pour notre problème, nous avons choisi deux algorithmes que nous avons jugé pertinents pour ce dernier. Deux algorithmes qui viennent de deux familles différentes de ce genre. Le premier fait partie de la famille **méta-heuristiques à parcours** et **méta-heuristiques à population**. Ces deux algorithmes, comme la plupart des algorithmes de ce genre, se sont inspirés des phénomènes naturels.

V.1 Algorithme recuit

L'algorithme recuit, qui fait parti des méta-heuristiques à parcours, s'inspire du recuit des métaux afin de modifier les caractéristiques de ces derniers. Le procédé consiste à

chauffer le métal à une température précise et ensuite le refroidir d'une manière contrôlé. Cela nous aide à contourner certaines contraintes physiques.

Dans notre cas, ce procédé peut nous aider à ne pas rester coincé dans un minimum/maximum local. En effet, nous parcourrons l'espace des solutions afin de trouver le plus petit coût. À cause de ça, si on fait un parcours normal comme "hill climbing", on peut se trouver proche d'un minimum local qui n'est pas global et ainsi, ne pas avoir des perfomances souhaitées. Cependant, il faut faire attention aux températures très élevées, car, elles peuvent nous éloigner de la solution optimale.

Nous devons aussi savoir contrôler le refroidissement. Un refroidissement très rapide peut causer des résultats loin de la solution optimale, et un refroidissement très lent peut prolonger le comportement chaotiques des températures élevées et causer un temps de calcule très élevé.

Recuit simple

Dans le recuit simple, on a un seul "chercheur" pour la solution. L'algorithme initial pour un refroidissement exponentiel de paramètre λ et une température initiale T_{init} et une température limite T_{limit} est :

1. On part d'une solution aléatoire et on la nomme *best*. On pose $T = T_{init}$
2. On génère un voisin aléatoire nommé *voisin*.
3. On évalue les deux
4. Si le voisin a une meilleure évaluation, ici meilleure c'est plus petit, on pose $prob = 1$ et on va à la ligne 6.
5. Sinon, on pose $prob = e^{-\frac{(evaluation(voisin)-evaluation(best))}{T}}$
6. On tire un nombre aléatoire nommé *rand* entre 0 et 1 suivant la loi uniforme.
7. Si $rand < prob$ alors $best = voisin$
8. T = λT
9. Si $T < T_{limit}$ on part à la ligne 2 sinon on arrête et on renvoie *best*

Notez que dans le terme de l'exponentielle, on a aussi la constante de Boltzmann au dénominateur qui est égal à 0 dans notre algorithme. Remarquons que la température a une influence direct sur *prob* et si on a une température élevé, *prob* va être proche de 1, donc, on a une forte probabilité de changer *best* même si l'évaluation de *voisin* n'est pas meilleure. Cela explique les fluctuations à températures élevée et la nécessité de décroître la température rapidement vers un niveau stable. C'est pour cette raison que nous avons choisi une décroissance exponentielle de la température qui nous assure ce dernier critère, et en même temps, laisse un peu de temps à l'algorithme pour converger vers le minimum le plus proche.

Pour générer un voisin aléatoire, nous prenons une indice aléatoirement et on change i-ème valeur, de sorte que, si c'est 0, on met 1, sinon, on met 0.

Pour évaluer les solutions, on prend déjà la somme des poids dans la section modélisation, de plus comme on ne veut pas de solutions invalides, on les pénalise avec un coefficient grand nommé *malus*. Dans notre algorithme, nous avons pris $malus = 500$ afin de mettre une différence remarquable entre les solutions valides et invalides. Les so-

lutions invalides sont les solutions qui ne sont pas des arbre ou elles ne contiennent pas les terminaux ou ne sont pas connexes. Autrement dit, on doit compter le nombre de composants connexes dans le graphe engendré par la solution et le nombre de terminaux absent dans ce même graphe. On peut s'en passer des cycles, car, ils sont déjà pénalisés dans la somme des poids. Pour le nombre de terminaux absents, on a doublé le malus, car, avoir des terminaux absents dans la solution est très grave. On a donc avec *absents* qui représente les terminaux absents et *connexes* qui représente les nombre de composantes connexes :

$$evaluation = 1000 * absents + 500 * (connexes - 1) + \sum_{e \in E'} w(e)$$

Pour l'algorithme initial, nous avons pris $\lambda = 0.99$. Cependant, afin de permettre une convergence vers le minimum, dans la version finale nous changeons λ à partir d'une température T_{seuil}. Grâce à ce changement, on a un meilleur résultat, mais, on rallonge le temps de calcul. Voici la version finale :

1. On part d'une solution aléatoire et on la nomme *best*. On pose $T = T_{init}$
2. On génère un voisin aléatoire nommé *voisin*.
3. On évalue les deux
4. Si le voisin a une meilleure évaluation, ici meilleure c'est plus petit, on pose $prob = 1$ et on va à la ligne 6.
5. Sinon, on pose $prob = e^{-\frac{(evaluation(voisin) - evaluation(best))}{T}}$
6. On tire un nombre aléatoire nommé *rand* entre 0 et 1 suivant la loi uniforme.
7. Si $rand < prob$ alors $best = voisin$
8. T = λT
9. Si $T < T_{seuil}$ alors $\lambda = \lambda_{alternative}$
10. Si $T < T_{limit}$ on part à la ligne 2 sinon on arrête et on renvoie *best*

Recuit multiple

Seul on va plus vite, ensemble, on va plus loin ! Cette expression résume bien l'intérêt d'avoir plusieurs chercheurs. Avec un temps de calcul un peu plus long, on peut avoir des résultats plus intéressants que recuit simple. Pour cela, au lieu de partir d'une solution aléatoire, on part de n solutions aléatoires qu'on met dans une liste nommée *bests*. On utilise la même évaluation que recuit simple.

1. On part de n solutions aléatoires et on la nomme *bests*. On pose $T = T_{init}$
2. On fait les étapes 2 à 9 de recuit simple pour chaque membre de *bests*
3. Si $T < T_{limit}$ on part à la ligne 2 sinon on arrête et on renvoie la solution qui a la meilleure évaluation parmi les membres de *bests*.

Nous allons voir dans la section résultats la différences entre ces deux versions.

V.2 Algorithme génétique

Faisant partie de la famille des méta-heuristiques à populations, il s'inspire aussi d'un phénomène naturel, l'évolution et la sélection naturelle. On part encore d'une solution aléatoire et on crée une population de solutions avec une méthode "génération". Puis, à

l'aide de l'évaluation vu dans recuit, on sélectionne les meilleures. On répète ce procédé un nombre défini de fois. On garde la même représentation de solution.

Génération

Pour générer de nouvelles solutions, on s'inspire de la reproduction naturelle. D'abord on croise deux solutions, ensuite, on prend une indice aléatoire et on mute la valeur à cette indice, *i.e.*, on met 0 si c'est 1, sinon, 1. Cependant, comme nos simulations nous l'ont montrer, cette méthode peut s'avérer lente en convergence vers la solution optimale, car, elle ne génère pas assez de solutions. Il faut donc accélérer le processus. Alors, au lieu de muter seulement les nouveaux nés, on génère aussi une copie mutée de toutes les solutions de la populations. Ainsi, on évaluent beaucoup plus de solutions à chaque fois et on a une meilleure chance pour trouver la meilleure solution.

Pour le croisement, on croise systématiquement les deux meilleures solutions. Encore une fois, on s'est inspiré de la nature, parce que dans beaucoup d'espèces, seul le couple alpha ont le droit de se reproduire. On génère un premier enfant qui a la première moitié du parent 1 et la deuxième moitié du parent 2. Pour le deuxième enfant, on fait l'inverse, *i.e.*, la première moitié vient du parent 2 et la deuxième du parent 1. Mais nous ne nous sommes pas contenté du couple alpha et à chaque reproduction, nous prenons aléatoirement deux solutions et on les croise. Grâce à cela, on peut avoir une population plus variée.

Sélection

Pour la sélection, on doit d'abord choisir le nombre maximum de solutions qu'on veut garder nommée m. On doit pas prendre m très petit, sinon, on risque de ne pas générer des solutions variées, et ainsi, ne pas converger vers la solution optimale. On ne doit pas prendre m très grand, sinon, on risque d'augmenter drastiquement le temps de calcul. Nous avons choisi $m = 15$ pour avoir une population assez grande afin évaluer un grand nombre de solutions, et assez petit pour être raisonnable en temps de calcul. Ce nombre a été trouver par tâtonnement à l'aide des simulations.

Algorithme

Soit n le nombre d'itération.

1. On part d'une solution aléatoire et on la nomme *best*.
2. On génère une nouvelle génération
3. On garde les m meilleures.
4. Si le meilleur de la population est meilleur que *best*, alors, on remplace *best*.
5. On répète n fois à partir de 2.
6. On renvoie *best*

VI Résultats et performances